Refining definitions with unknown opens using XSB for IDP³

Joachim Jansen, Gerda Janssens

Department of Computer Science, KU Leuven
joachim.jansen, gerda.janssens@cs.kuleuven.be

Abstract. FO(·)IDP is a declarative modeling language that extends first-order logic with inductive definitions, partial functions, types and aggregates. Its model generator IDP³ grounds the problem into a low-level (propositional) representation and consequently uses a generic solver to search for a solution. Recent work introduced a technique that evaluates all definitions that depend on fully known information before the grounding step. In this paper, we extend this technique, which allows us to refine the interpretation of defined symbols when they depend on information that is only partially given instead of completely given. We use our existing transformation of FO(·)IDP definitions to Tabled Prolog rules and extend it to support definitions that depend on information that is possibly partially unknown. In this paper we present an algorithm that uses XSB Prolog to evaluate these rules in such a way that we achieve the most precise possible refinement of the defined symbols. Experimental results show that our technique derives extra information for the defined symbols.

1 Introduction

Recent proposals for declarative modeling use first-order logic as their starting point. Examples are Enfragmo [1] and FO(·)IDP, the instance of the FO(·) family that is supported by IDP³, the current version of the IDP Knowledge Base System [6]. FO(·)IDP extends first-order logic (FO) with inductive definitions, partial functions, types and aggregates. IDP³ supports model generation and model expansion [11, 4] as inference methods.

IDP³ supports these inference methods using the ground-and-solve approach. First the problem is grounded into an Extended CNF (ECNF) theory. Next a SAT-solver is used to calculate a model of the propositional theory. The technique that is presented in this paper is to improve the efficiency and robustness of the grounding step. One of the problems when grounding is the possible combinatorial blowup of the grounding. A predicate \( p(x_1, x_2 \ldots x_n) \) with \( s \) as the size of the domain of its arguments has \( s^n \) possible instances. A grounding that has to represent all these possible instances is therefore possibly very large. Most Answer Set Programming (ASP) systems solve this problem by using semi-naive bottom-up evaluation [8, 9] with optimizations. On a high level IDP³ uses three
techniques to manage the complexity of the grounding process: definition evaluation [10], Lifted Unit Propagation (LUP) [15] and Grounding With Bounds (GWB) [16].

Our previous work [10] is a pre-processing step that calculates in advance the two-valued interpretations for defined predicates that depend on fully known information. We call such defined predicates input * predicates. The definitions of these predicates and the information on which they depend are translated into a XSB Prolog [12] program that tables the defined input * predicates, supporting the well-founded semantics [14, 13]. This Tabled Prolog program is then queried to retrieve the atoms for which the tabled predicates are true. The input structure (the initially given partial structure) is extended with the calculated information. The input * predicates become completely known: they are true for the tabled atoms and false for all the other instances. As a result, definitions of the input * predicates are no longer needed and they are removed from the problem specification.

Lifted Unit Propagation (LUP) is another pre-processing step that further refines the partial structure. LUP propagates knowledge about true and false atoms in the formulas of the FO(·)IDP theory. For the definitions of the FO(·)IDP theory LUP uses an approximation by propagating on the completion of the definitions. The method of this paper is an alternative for using LUP on the completion of the definitions. We extend our existing pre-processing step [10] to be able to refine the interpretation of defined predicates in the partial structure when the predicates depend on information that is only partially given. This extension can then be used as an alternative to executing LUP on the completion of definitions. Our method uses XSB to compute the atoms (instances of the predicate) that are true and others that are unknown. The computed atoms are used to refine the partial structure. Moreover, XSB’s support for the well-founded semantics makes atoms false when XSB detects unfoundedness. This detection of unfoundedness is not present in the approach that uses LUP on the completion of definitions to refine them.

Grounding With Bounds (GWB) uses symbolic reasoning when grounding subformulas to derive bounds. Because GWB uses the input structure, it can benefit from the extra information that is inferred thanks to the refinement done by LUP. Using this extra information, possibly tighter bounds can be derived. Therefore it is beneficial to refine the input structure as much as possible before grounding (using GWB). Because of this, we will measure the effectiveness of the discussed methods by how much they are able to refine the input structure. The actual grounding process that will benefit from this refined structure is considered out of scope for this paper.

Our contribution is a new way to perform lifted propagation for definitions. Experimental results compare the new technique with the old one of performing LUP for the completion of the definition.

In Section 2 we introduce IDP 3 and FO(·). Section 3 explains our approach using an example. In Section 4 we describe the extensions to the transformation to Tabled Prolog rules and the workflow of our interaction with XSB. Section 5
presents the high-level algorithm that is used to refine all defined symbols as much as possible using the previously mentioned XSB interaction. In Section 6 we present experimental results. Section 7 contains future work and concludes.

2 Terminology and Motivation

2.1 The FO(·)IDP language

We focus on the aspects of FO(·)IDP that are relevant for this paper. More details can be found in [6] and [2], where one can find several examples. An FO(·)IDP model consists of a number of logical components, a.o. vocabularies, structures, and theories.

A vocabulary declares the symbols to be used.

A structure is used to specify the domain and data; it provides an interpretation of the symbols in the vocabulary. The interpretation of a symbol specifies for this symbol which atoms (instances) are true, unknown, and false. Interpretations that contain elements that are unknown are also called a partial (or three-valued) interpretation. Otherwise, the interpretation is said to be two-valued.

A theory consists of FO(·)IDP formulas and definitions. An FO(·)IDP formula differs from FO formulas in two ways. Firstly, FO(·)IDP is a many-sorted logic: every variable has an associated type and every type an associated domain. Moreover, it is order-sorted: types can be subtypes of others. Secondly, besides the standard terms in FO, FO(·)IDP formulas can also have aggregate terms: functions over a set of domain elements and associated numeric values which map to the sum, product, cardinality, maximum or minimum value of the set.

An FO(·)IDP definition is a set of rules of the form \( \forall \bar{x} : p(\bar{x}) \leftarrow \phi[\bar{x}] \), where \( \phi[\bar{x}] \) is an FO(·)IDP formula. We call \( p(\bar{x}) \) the head of the rule and \( \phi[\bar{x}] \), the body of the rule. The defined symbols of a theory are the symbols that appear in a head of any rule. The other symbols, which appear only in bodies of definitions are the open symbols. We remind the reader that previous work [10] describes a transformation of FO(·)IDP definitions into Tabled Prolog rules. This includes a transformation of the interpretation of the open symbols to (Tabled) Prolog facts.

2.2 The IDP³ system

IDP³ is a Knowledge Base System [6], meaning it supports a variety of problem-solving inferences. One of these inferences is model expansion. The model expansion of IDP³ extends a partial structure (an interpretation) into a two-valued structure that satisfies all constraints specified by the FO(·)IDP model. Formally, the task of model expansion is, given a vocabulary \( V \), a theory \( T \) over \( V \) and a partial structure \( S \) over \( V \) (at least interpreting all types), to find a two-valued structure \( M \) that satisfies \( T \) and extends \( S \), i.e., \( M \) is a model of the theory and the input structure \( S \) is a subset of \( M \).
As mentioned before, IDP\(^3\) uses the ground-and-solve approach. It grounds the problem and then uses the solver MiniSAT(ID) [3, 5], based on the solver MiniSAT [7].

There are three techniques that IDP\(^3\) uses to optimise its grounding process: definition evaluation [10], Lifted Unit Propagation (LUP) [15] and Grounding With Bounds (GWB) [16].

Our previous work [10] introduces a pre-processing step that reduces the IDP\(^3\) grounding by calculating some definitions in advance. We calculate the two-valued interpretations for defined predicates that depend on completely known information. We transform the relevant definition into Tabled Prolog rules, we add the relevant fragment of the input structure as Prolog facts, and we query XSB for the desired interpretation. We use the computed atoms to complete the two-valued interpretation for the defined symbols. The definitions are no longer needed and can be removed from the theory.

LUP can most easily be explained based on what SAT solvers do. Most SAT solvers start by performing Unit Propagation (UP) on the input to derive new information about the search problem. LUP is designed to refine the input structure using unit propagation, but on the FO($)\text{Top} formula instead of on the ground representation, which is why it is called “lifted”. It is important to note that LUP only refines the structure with respect to the formulas and not w.r.t. the definitions. To resolve this, LUP is executed for the completion of the definitions, but this is an approximation of what can be derived from definitions. In this paper, we extend the technique used to evaluate definitions to perform lifted propagation on definitions that have opens with a three-valued interpretation. This extension can then be used as an alternative to executing LUP on the completion of definitions.

GWB uses symbolic reasoning when grounding subformulas. Given the input structure, it derives bounds for certainly true, certainly false and unknown for quantified variables over (sub)formulas. Consequentially, since GWB uses the structure, it can benefit from the extra information that is inferred thanks to the refinement done by LUP. Using this extra information, possibly tighter bounds can be derived.

3 Example of refining structures

The example shown in Figure 1 expresses a reachability problem for colored nodes using undirected edges. We use this example to illustrate some of the concepts.

The theory \(T\) in the example contains one formula and two definitions: one definition defines the symbol \(\text{uedge}/2\) and the other definition defines \(\text{reach}/2\). We abuse notation and use “\(\text{uedge}/2\) definition” to denote the definition defining the \(\text{uedge}/2\) symbol. The \(\text{uedge}/2\) definition has only one open symbol: \(\text{edge}/2\). Because \(\text{edge}/2\) has a two-valued interpretation, our original method [10] is applicable, so we perform definition evaluation for the \(\text{uedge}/2\) definition. The calculated interpretation for \(\text{uedge}/2\) can be seen in \(S_2\), depicted in Figure 2.
Fig. 1. An IDP³ problem specification example. The notation color<ct> and color<cf> is used to specify which elements are certainly true, respectively certainly false for the color(node,color) relation in structure $S$. Tuples that are in neither of these specifications are unknown.

The reach/2 definition has three open symbols: start/1, uedge/2, and color/2. Because color/2 has a three-valued interpretation, we cannot perform definition evaluation for the reach/2 definition. This concludes what can be done with regards to definition evaluation and we proceed with the theory $T^2$ and $S^2$ as depicted in Figure 2. For compactness, $S^2$ only shows symbols for which the interpretation has changed with regards to $S$. The vocabulary remains the same as in Figure 1.

Next, we can perform Lifted Unit Propagation for the formula in $T^2$. This formula expresses that when a node is RED, it cannot be BLUE and vice versa. Since the structure $S^2$ specifies that node 3 cannot be RED, we derive that node 3 has to be BLUE. In the same manner we can derive that node 2 cannot be BLUE. This results in structure $S^3$ as depicted in Figure 2. The theory remains the same as in Figure 2.

This leaves us with the reach/2 definition to further refine $S^3$. There are two approaches to performing lifted propagation on this definition: first we can perform LUP on the completion of the reach/2 definition or alternatively, we use the new method introduced in this paper. Structure $S^4$ in Figure 4 shows what can be derived using the existing LUP method on the completion of the reach/2 definition, which is the following equivalence:

$$\forall y \ c : \ reach(y, c) \leftrightarrow start(y) \vee \exists x : (reach(x, c) \land uedge(x, y) \land color(y, c)).$$

Note that in $S^4$ node 1 is reachable using BLUE as well as RED because the first rule in the reach/2 definition says the starting node is always reachable.
theory T2 : V { 
{ reach(x,c) ← start(x). 
reach(y,c) ← reach(x,c) ∧ uedge(x,y) ∧ color(y,c). } 
∀x: color(x,RED) ⇔ ¬color(x,BLUE). 
} 
structure S2 : V { 
uedge = { 1,2; 1,3; 2,1; 2,4; 3,1; 3,4; 3,5; 4,2; 
        4,3; 5,3; 6,6 } 
} 

Fig. 2. The theory and structure after performing definition evaluation on T and S. The interpretation of node, start/1, color/2 and edge/2 remains the same as in S.

structure S3 : V { 
color<ct> = {2,RED; 3,BLUE} color<cf> = {2,BLUE; 3,RED} 
} 

Fig. 3. The structure after performing LUP on the formula in T2 using S2. The interpretation of node, start/1, edge/2 and uedge/2 remains the same as in S2.

with all colors. Also note that S4 specifies that reach(5, RED) is false because there is no edge from a RED reachable node to 5.

structure S4 : V { 
reach<ct> = { 1,BLUE; 1,RED; 2,RED; 3,BLUE } 
reach<cf> = { 2,BLUE; 3,RED; 5,RED } 
} 

Fig. 4. The structure after LUP on the completion of the reach/2 definition using S3. The interpretation of all other symbols remains the same as in S3.

Structure S5 in Figure 5 shows the result after executing definition refinement. Structure S5 is more refined than structure S4, since it specifies the atoms (6,RED) and (6,BLUE) to be false, whereas these atoms are unknown in structure S4. These atoms can be derived to be false because they form an unfounded set under the Well-Founded Semantics [14]. An unfounded set is a set of atoms that only have a rule making them true that depends on themselves. The definition, which is shown below for y = 6 and x = 6, illustrates that the above derived atoms are an unfounded set. The first rule of the definition is not applicable since start(6) is false. The second rule shows that the truth value of
**structure** S5 : V {
reach <ct> = { 1, BLUE; 2, RED; 3, BLUE }
reach <cf> = { 2, BLUE; 3, RED; 5, RED; 6, RED; 6, BLUE }
}

Fig. 5. The structure after definition refinement on the reach/2 definition using S3. The interpretation of all other symbols remains the same as in S3

reach(6, c) depends on reach(6, c) itself.

\[
\begin{cases}
\text{reach}(6, c) \leftarrow \text{start}(6). \\
\text{reach}(6, c) \leftarrow \text{reach}(6, c) \land \text{edge}(6, 6) \land \text{color}(6, c).
\end{cases}
\]

This concludes our example of the different ways of performing lifted propagation to refine the input structure. Our next section presents the changes we had to make to our original approach for evaluation definition to extend it for definition refinement. Section 4.3 contains the complete interaction between the XSB interface and IDP that is needed to perform the above definition refinement for the reach/2 definition.

4 Updating the XSB interface

Our new method differs only in a few ways from our original technique’s usage of XSB [10]. The transformation of the inductive definitions to an XSB program does not need to change. Here we discuss the necessary extensions:

- Provide support for translating the interpretation of symbols that are not completely two-valued to XSB.
- Update the interaction with XSB to also query the possible unknown answers for the queried definition.

4.1 Translating unknown opens

Open symbols can now have an interpretation for which the union of the certainly true and certainly false tables does not contain all elements. Therefore we need to provide a translation for the unknown elements in the interpretation of an open symbol. We illustrate this using the following example: \( q(x) \) is an open symbol and the type of \( x \) ranges from 1 to 5. Say \( q(x) \) is known to be true for \{1, 2\} and known to be false for \{4, 5\}. As a result, the truth value for \( q(3) \) is not known. The open symbol \( q(x) \) for the above interpretation will be represented in XSB as follows, given that \text{xsb\_q(X)} is the corresponding symbol present in the XSB program for \( q(x) \):

\[
\begin{align*}
\text{:- table xsb\_q/1.} \\
\text{xsb\_q(1).}
\end{align*}
\]
Calling \texttt{xsb\_q(X)} results in \( X = 1 \), \( X = 2 \), and \( X = 3 \), with \( X = 3 \) being annotated as "undefined". This is because XSB detects the loop over negation for \( X = 3 \). Note the use of \texttt{tnot/1} instead of the regular \texttt{not/1} to express negation. This is because \texttt{tnot/1} expresses the negation under the Well-Founded Semantics for tabled predicates, whereas \texttt{not/1} expresses Prolog's negation by failure.

### 4.2 Updating the interaction with XSB

We explain the change in interaction using an example. Say we are processing a definition that defines symbol \( p(x) \). Let \texttt{xsb\_p(X)} be the corresponding symbol present in the XSB program. The original interaction between XSB and IDP\(^3\) [10] queries XSB with

\[
\texttt{:- call\_tv(xsb\_p(X), true).}
\]

which computes all values of \( X \) for which \texttt{xsb\_p(X)} is \texttt{true} and retrieves the table of results, which we shall call \( t_t \). Next, we change the interpretation of \( p(x) \) in the partial structure into a two-valued one in which the atoms in the table \( t_t \) are \texttt{true} and all the others are \texttt{false}.

The new XSB interface uses the same query as above and additionally queries XSB with

\[
\texttt{:- call\_tv(xsb\_p(X), undefined).}
\]

which computes all values of \( X \) for which \texttt{xsb\_p(X)} is annotated as \texttt{undefined} and retrieves the table of results, which we shall call \( t_u \). Next, we change the interpretation of \( p(x) \) in the partial structure into a three-valued one in which the atoms in the table \( t_t \) are \texttt{true}, the atoms in table \( t_u \) are \texttt{unknown} and all the others are \texttt{false}.

### 4.3 Example of a complete run

This section gives a complete overview of all the actions for performing definition refinement on the \texttt{reach/2} definition from Section 3. First, the definition is translated into an XSB program:

\[
\texttt{:- set\_prolog\_flag(unknown, fail).}
\]

\[
\texttt{:- table xsb\_reach/2.}
\]

\[
\texttt{xsb\_reach(X, C) :- xsb\_start(X), xsb\_color\_type(C).
}\]

\[
\texttt{xsb\_reach(Y, C) :- xsb\_reach(X, C), xsb\_uedge(X, Y), xsb\_color(Y, C).
}\]
And the structure is also translated into a corresponding XSB program:

```
xsb_start(1).

xsb_color_type(xsb_RED).
xsb_color_type(xsb_BLUE).

xsb_uedge(1, 2).
xsb_uedge(1, 3).
xsb_uedge(2, 1).
xsb_uedge(2, 4).
xsb_uedge(3, 1).
xsb_uedge(3, 4).
xsb_uedge(3, 5).
xsb_uedge(4, 2).
xsb_uedge(4, 3).
xsb_uedge(5, 3).
xsb_uedge(6, 6).

:- table xsb_color/2.
xsb_color(1, xsb_RED) :- undef.
xsb_color(1, xsb_BLUE) :- undef.
xsb_color(2, xsb_RED).
xsb_color(3, xsb_BLUE).
xsb_color(4, xsb_RED) :- undef.
xsb_color(4, xsb_BLUE) :- undef.
xsb_color(5, xsb_RED) :- undef.
xsb_color(5, xsb_BLUE) :- undef.
xsb_color(6, xsb_RED) :- undef.
xsb_color(6, xsb_BLUE) :- undef.

:- table undef/0.
undef :- tnot(undef).
```

These two programs are then loaded, along with some utility predicates. Next, we query XSB with the following queries:

```
| ?- call_tv(xsb_reach(X, Y), true).
X = 3, Y = xsb_BLUE;
X = 2, Y = xsb_RED;
X = 1, Y = xsb_BLUE;
X = 1, Y = xsb_RED;
no
| ?- call_tv(xsb_reach(X, Y), undefined).
X = 5, Y = xsb_BLUE;
X = 4, Y = xsb_BLUE;
X = 4, Y = xsb_RED;
```
As a final step, the interpretation for \textit{reach}/2 is changed so that it is \textit{true} for \{(3, BLUE) (2, RED) (1, BLUE) (1, RED)\} and that it is \textit{unknown} for \{(5, BLUE) (4, BLUE) (4, RED)\}, and \textit{false} for everything else. This is depicted in Figure 5.

5 Lifted Propagation

The previous section explains how we construct an interface to XSB to retrieve a refined interpretation for the defined symbols in a single definition. Algorithm 1 shows an algorithm that uses this XSB interface to refine a structure as much as possible when there are multiple definitions in a theory. For the scope of this algorithm, the XSB interface is called as if were a subroutine (called \textsc{XSB-interface}). We maintain the set of definitions that need to be processed as \textit{Set}. Initially, \textit{Set} contains all definitions and until \textit{Set} is empty, we take one definition from it and process it using the XSB interface. The most important aspect of the presented algorithm is in line 12, where definitions that may have been processed before, but have an open symbol that was “updated” by processing another definition, are put back into \textit{Set} to be processed again.

\begin{algorithm}
\textbf{input}: A structure \(S\) and a set \(\Delta\) of definitions in theory \(T\)  
\textbf{output}: A new structure \(S'\) that refines \(S\) as much as possible using \(\Delta\)  
\begin{algorithmic}[1]
\STATE \(\text{Set} \leftarrow \Delta\)
\WHILE{\text{Set is not empty}}
\STATE \(\delta \leftarrow \text{an element from Set}\)
\STATE \(\text{XSB-interface} (\delta, S)\)
\IF{inconsistency is detected}
\STATE return an inconsistent structure
\ENDIF
\STATE \(\Sigma \leftarrow \text{The symbols for which the interpretation has changed}\)
\FOR{\(\delta'\) in \(\Delta\)}
\IF{\(\delta'\) has one of \(\Sigma\) in its opens}
\STATE add \(\delta'\) to \(\text{Set}\)
\ENDIF
\ENDFOR
\STATE remove \(\delta\) from \(\text{Set}\)
\end{algorithmic}
\end{algorithm}

Algorithm 1: Lifted Propagation for multiple definitions

On line 5 we need to detect when an inconsistency arises from processing a definition. On line 9 we retrieve all symbols for which the interpretation has changed by processing definition \(\delta\). Since these features were not mentioned in the previous section we shortly explain here how these can be achieved. When
the XSB interface processes a definition (say, XSB-INTERFACE \((\delta, S)\) is called), it does not use the interpretation of the defined symbols in \(\delta\) in \(S\) for any of its calculations. We use \(I_\sigma\) to denote the interpretation of defined symbol \(\sigma\) in structure \(S\). XSB calculates a “new” interpretation for every defined symbol \(\sigma\) in \(\delta\), which we will call \(I'_\sigma\). If the number of true or the number of false atoms in \(I_\sigma\) and \(I'_\sigma\) differ, XSB has changed the interpretation of symbol \(\sigma\) and this symbol will be present in \(\Sigma\) as displayed in line 9. If there is an atom that is true in \(I_\sigma\) and false in \(I'_\sigma\), or vice versa, there is inconsistency and the check on line 5 will succeed.

A possible point of improvement for this algorithm is the selection done in line 3. One could perform a dependency analysis and stratify the definitions that have to be refined. In this way, the amount of times each definition is “processed” is minimized. This stratification is ongoing work.

A worst case performance for the proposed algorithm is achieved when there are two definitions that depend on each other, as given in the following example:

\[
\begin{align*}
P(0). \\
P(x) &\leftarrow Q(x - 1). \\
Q(x) &\leftarrow P(x - 1).
\end{align*}
\]

If we start with processing the \(P/1\) definition, we derive \(P(0)\). Processing \(Q/1\) then leads to deriving \(Q(1)\). Since the interpretation of \(Q/1\) changed and it is an open symbol of the \(P/1\) definition, the \(P/1\) definition has to be processed again. This continues for as many iterations as there are elements in the type of \(x\). Since every call to the XSB interface for a definition incurs inter-process overhead, this leads to a poor performance. This problem can be alleviated by detecting that the definitions can safely be joined together into a single definition. The detection of joining definition to improve the performance of the proposed algorithm is part of future work.

6 Experimental evaluation

In this section we evaluate our new method of refining definitions by comparing it to its alternative: performing Lifted Unit Propagation (LUP) on the completion of the definitions. We will refer to our new method for Definition Refinement as “the DR approach”. In the IDP$^3$ system, there are two ways of performing LUP on a structure: using an Approximating Definition (AD) \([4]\) or using Binary Decision Diagrams (BDD) \([15]\). We will refer to these methods as “the AD approach” for the former and “the BDD approach” for the latter. The AD approach expresses the possible propagation on SAT level using an IDP$^3$ definition. This definition is evaluated to derive new true and false bounds for the structure. Note that this approximating definition is entirely different from any other possible definitions originally present in the theory. The BDD approach works by creating Binary Decision Diagrams that represent the formulas (in this case the
formulas for the completion of the definitions) in the theory. It works symbolically and is approximative: it will not always derive the best possible refinement of the structure. The AD approach on the other hand is not approximative.

Table 1 shows our experiment results for 39 problems taken from past ASP competitions. For each problem we evaluate our method on 10 or 13 instances. The problem instances are evaluated with a timeout of 300 seconds. We present the following information for each problem, for the DR approach:

- $s^{DR}$ The number of runs that succeeded
- $t_{avg}^{DR}$ The average running time (in seconds)
- $t_{max}^{DR}$ The highest running time (in seconds)
- $a_{avg}^{DR}$ The average number of derived atoms

The same information is also given for the BDD and the AD approach, with the exception that $a_{avg}^{BDD}$ and $a_{avg}^{AD}$ only take into account runs that also succeeded for the DR approach. This allows us to compare the number of derived atoms, since it can depend strongly on the instance of the problem that is run.

Comparing the DR approach with the AD approach, one can see that the DR approach is clearly better. The AD approach fails to refine the structure for even a single problem instance for 20 out of the 39 problems. When both the DR and the AD approaches do succeed, AD derives as much information as the DR approach. One can conclude from this that there is no benefit to using the AD approach over the DR approach. Moreover, the DR approach is faster in most of the cases.

Comparing the DR approach with the BDD approach is less straightforward. The BDD approach has a faster average and maximum running time for each of the problems. Additionally, for 11 out of the 39 problems the BDD approach had more problem instances that did not reach a timeout. These problems are indicated in bold in the $s^{BDD}$ column. For some problems the difference is small, as for example for the ChannelRouting problem where average running times are respectively 7.16 and 5.67. For other problems however, the difference is very large, as for example for the PackingProblem problem where average running times are respectively 199.53 with 7 timeouts and 0.1 with 0 timeouts. Although the BDD approach is clearly faster, there is an advantage to the DR approach: for 8 problems, it derives extra information compared to the BDD approach. These instances are indicated in bold in the $a_{avg}^{DR}$ column. This difference is sometimes small (80 vs. 8 for SokobanDecision) and sometimes large (25119 vs. 0 for Tangram). This shows that there is an advantage to using our newly proposed DR approach.

There is one outlier, namely the NoMystery problem in which more information is derived with the BDD approach than by the DR approach. This is because DR does lifted propagation in the “direction” of the definition: for the known information of the body of a rule, try to derive more information about the defined symbol. However, sometimes it is possible to derive information about elements in the body rules using information that is known about the head of the rule. Since LUP performs propagation along both directions and DR only along the
first one, it is possible that LUP derives more information. As one can see in the experiment results, it is only on very rare occasions (1 problem out of 39) where this extra direction makes a difference. Integrating this other direction of propagation into the DR approach is ongoing work.

<table>
<thead>
<tr>
<th>Problem Name</th>
<th>Definition Refinement</th>
<th>Binary Decision Diagram</th>
<th>Approximating Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>15Puzzle</td>
<td>0.00 5.3 6.24 482</td>
<td>0.00 0.07 0.08 256</td>
<td>0.00 0.0 - 0.0</td>
</tr>
<tr>
<td>Binomial</td>
<td>0.00 7.16 12.56</td>
<td>0.00 5.67 10.44</td>
<td>0.00 5.3 10.01 0</td>
</tr>
<tr>
<td>ClusterRouting</td>
<td>0.00 5.62 7.28</td>
<td>0.00 4.58 6.1</td>
<td>0.00 4.23 5.35 0</td>
</tr>
<tr>
<td>ConnectedDominatingSet</td>
<td>0.00 0.39 1.13</td>
<td>0.00 0.01 0.02</td>
<td>0.00 10.15.79 129.04 0</td>
</tr>
<tr>
<td>EdgeMatching</td>
<td>0.00 4.46 6.35</td>
<td>0.00 0.28 0.37</td>
<td>0.00 1.69 2.68 0</td>
</tr>
<tr>
<td>GraphPartitioning</td>
<td>0.00 0.27 0.48</td>
<td>0.00 0.02 0.02</td>
<td>0.00 13.14 45.73 0</td>
</tr>
<tr>
<td>HamiltonianPath</td>
<td>0.00 1.43 2.12</td>
<td>0.00 0.02 0.02</td>
<td>0.00 0 - 0</td>
</tr>
<tr>
<td>HierarchicalOrdering</td>
<td>0.00 9.11 89.86</td>
<td>0.00 6.59 92.12</td>
<td>0.00 6.42 63.24 0</td>
</tr>
<tr>
<td>MaxClique</td>
<td>0.00 3.2 7.08</td>
<td>0.00 2.38 4.03</td>
<td>0.00 6.05 27.65 45.34</td>
</tr>
<tr>
<td>SetCover</td>
<td>0.00 0.01 0.05</td>
<td>0.00 0.01 0.01</td>
<td>0.00 0.03 0.04 0</td>
</tr>
<tr>
<td>TravellingSalesman</td>
<td>0.00 0.51 0.71</td>
<td>0.00 0.06 0.06</td>
<td>0.00 0.72 0.84 73</td>
</tr>
<tr>
<td>WeightedSudokuDominatingSet</td>
<td>0.00 0.03 0.04</td>
<td>0.00 0.02 0.03</td>
<td>0.00 0.04 0.06 6</td>
</tr>
<tr>
<td>WireRouting</td>
<td>0.00 1.25 2.26</td>
<td>0.00 0.12 0.18</td>
<td>0.00 0 - 0</td>
</tr>
<tr>
<td>GraphMatrixSplitterSet</td>
<td>0.00 - 0</td>
<td>0.00 10.13 2.28</td>
<td>0.00 0 - 0</td>
</tr>
<tr>
<td>FastFoodOptimalityCheck</td>
<td>1.00 18.43 18.43</td>
<td>30072 10.00 2.49</td>
<td>3.77 30072 0.00</td>
</tr>
<tr>
<td>SolitaireDecision</td>
<td>0.00 5.86 11.42</td>
<td>80.00 0.11 0.14</td>
<td>8.00 0 - 0</td>
</tr>
<tr>
<td>KnightTour</td>
<td>6.00 9.03 31.9</td>
<td>0.10 1.14 4.07</td>
<td>0.80 135.06 293.07 0</td>
</tr>
<tr>
<td>DynamicFlowScheduling</td>
<td>0.00 4.49 10.67</td>
<td>0.00 0.29 0.57</td>
<td>0.00 3.22 9.67 0</td>
</tr>
<tr>
<td>PackingPermutation</td>
<td>3.00 1909.32 43.46</td>
<td>170 10.00 0.13 13</td>
<td>17 0.00 0 - 0</td>
</tr>
<tr>
<td>Labyrinth</td>
<td>2.00 007.77 19.25 103533</td>
<td>0.00 0.62 1.12 104683</td>
<td>0.00 0 - 0</td>
</tr>
<tr>
<td>Numberlink</td>
<td>9.00 9.91 30.4</td>
<td>0.00 10.28 1.24</td>
<td>0.00 0 - 0</td>
</tr>
<tr>
<td>ReverseFolding</td>
<td>1.00 42.84 20.84</td>
<td>978 10.00 1.21</td>
<td>6 0.00 0 - 0</td>
</tr>
<tr>
<td>HanoiTower</td>
<td>0.00 28.42 52.87</td>
<td>20820 0.00 0.25</td>
<td>25042 0.00 0 - 0</td>
</tr>
<tr>
<td>MagicSquareSets</td>
<td>0.00 0.64 0.01</td>
<td>16589 0.00 0.12</td>
<td>15 0.00 0 - 0</td>
</tr>
<tr>
<td>AirplanePickup</td>
<td>0.00 4909 100078</td>
<td>3267 0.00 0.28</td>
<td>12472 0.00 0 - 0</td>
</tr>
<tr>
<td>PartnerUnits</td>
<td>0.00 13.43 14.97</td>
<td>7000 0.00 0.02</td>
<td>100 0.00 0 - 0</td>
</tr>
<tr>
<td>TinGates</td>
<td>13.13 37.84 41.38 205119</td>
<td>28.13 0.13 0.23</td>
<td>0.4 0.13 0 - 0</td>
</tr>
<tr>
<td>PreconditionFatters-Matching</td>
<td>0.00 2333.3 115.63</td>
<td>0.00 0.00 0.01</td>
<td>8 0.00 18.67 258.84 0</td>
</tr>
<tr>
<td>GeneralF-Graphs</td>
<td>0.00 0.01 0.02</td>
<td>0.00 0.00 0.01</td>
<td>0.00 0.03 0.04 0</td>
</tr>
<tr>
<td>Bottleneck-DisjointProblem</td>
<td>0.00 1.43 5.00</td>
<td>0.00 0.06 3.57</td>
<td>0.00 0.81 2.08 0</td>
</tr>
<tr>
<td>NoH-Interval</td>
<td>4.00 64.62 137.26 205894</td>
<td>0.10 0.57 1.3 20782</td>
<td>0.00 0 - 0</td>
</tr>
<tr>
<td>Solitaire</td>
<td>6.00 897.96 648 18254</td>
<td>0.10 0.11 0.12</td>
<td>8 0.00 0 - 0</td>
</tr>
<tr>
<td>Rooked-Defeat</td>
<td>0.00 - 0</td>
<td>0.00 2.68 3.16</td>
<td>0.00 0 - 0</td>
</tr>
<tr>
<td>Weighted-DisjointProblem</td>
<td>0.00 0.25 0.33</td>
<td>0.00 0.22 0.23</td>
<td>0.00 0.36 0.20 0</td>
</tr>
<tr>
<td>Incremental-Stocking</td>
<td>0.00 - 0</td>
<td>8 0.10 54.34 229.50 0</td>
<td>0.00 0 - 0</td>
</tr>
<tr>
<td>Visits-all</td>
<td>3.00 3.49 3.97</td>
<td>130 0.00 0.03 0.04</td>
<td>15 0.00 0 - 0</td>
</tr>
<tr>
<td>Graph-Coercing</td>
<td>0.00 0.12 0.15</td>
<td>0.00 0.02 0.03</td>
<td>0.00 0.02 0.08 0</td>
</tr>
<tr>
<td>LatinSquares</td>
<td>0.00 0.02 0.02</td>
<td>0.00 0.01 0.01</td>
<td>0.00 0.03 0.04 0</td>
</tr>
<tr>
<td>Sudoku</td>
<td>0.00 0.2 0.2</td>
<td>0.00 0.16 0.26</td>
<td>0.00 0.14 0.23 0</td>
</tr>
</tbody>
</table>

Table 1. Experiment results comparing Definition Refinement (DR) with the Binary Decision Diagram (BDD) and Approximating Definition (AD) approach

Our experiments show the added value of our DR approach, but also indicate that more effort should be put into this approach towards optimising runtime.

7 Conclusion

In this paper we described an extension to our existing preprocessing step [10] for definition evaluation to be able to refine the interpretation of defined predicates in the partial structure when the predicates depend on information that is only
partially given. Our method uses XSB to compute the atoms (instances of the predicate) that are true and others that are unknown. This method is an alternative for using LUP on the completion of the definitions. Because LUP for the completion of the definition is an approximation of what can be derived for that definition, our method is able to derive strictly more information for the defined symbols than the LUP alternative. The extra information that can be derived is the detection of unfounded sets for a definition. Because GWB uses the information in the structure to derive bounds during grounding, this extra derived information possibly leads to stricter bounds and an improved grounding.

Our experiments show the added value of our new method, but also indicate that it is not as robust in performance as LUP (using BDDs). This paper indicates two ways in which the performance of the proposed method might be improved:

- Perform an analysis of the dependencies of definitions and query them accordingly to minimize the number of times a definition is re-queried.
- Similar to the element above, try to detect when definitions can be joined together to minimize XSB overhead

These improvements are future work. Another part of future work is combining the new method for lifted propagation for definitions with the LUP for formulas in the theory. Combining these two techniques might lead to even more derived information, since the formulas might derive information that allows the definition the perform more propagation and vice versa.

References